A simple GPU-accelerated two-dimensional MUSCL-Hancock solver for ideal magnetohydrodynamics

Christopher M. Bard[a,*], John C. Dorelli[b,**]

[a] *University of Wisconsin, Madison*
[b] *NASA-GSFC*

Abstract

We describe our experience using NVIDIA's CUDA (Compute Unified Device Architecture) C programming environment to implement a two-dimensional second-order MUSCL-Hancock ideal magnetohydrodynamics (MHD) solver on a GTX 480 Graphics Processing Unit (GPU). Taking a simple approach in which the MHD variables are stored exclusively in the global memory of the GTX 480 and accessed in a cache-friendly manner (without further optimizing memory access by, for example, staging data in the GPU's faster shared memory), we achieved a maximum speed-up of ≈ 126 for a 1024^2 grid relative to the sequential C code running on a single Intel Nehalem (2.8 GHz) core. This speedup is consistent with simple estimates based on the known floating point performance, memory throughput and parallel processing capacity of the GTX 480.

Keywords: GPU, Magnetohydrodynamics

1. Introduction

The last several years has witnessed a dramatic increase in the use of Graphics Processing Units (GPUs) to accelerate scientific computing applications. The primary catalyst for this surge in GPU computing was NVIDIA's public release of the CUDA (Compute Unified Device Architecture) programming environment [1] in 2007. The introduction of CUDA provided the scientific programmer with easy access to the SIMT (Single Instruction Multiple Thread) architecture underlying modern NVIDIA GPUs (previously accessible only through cumbersome graphics APIs like OpenGL) via an intuitive extension of ANSI C. The result has been a proliferation of GPU-accelerated applications in such diverse areas as N-body simulation [2, 3], signal processing [4], molecular dynamics simulation [5, 6] and computational fluid dynamics (CFD) [7, 8]. Speedups reported in the literature depend on the application, but one can expect, on average, an order of magnitude performance increase "out of the box" using CUDA.

Explicit finite volume schemes are particularly amenable to GPU acceleration due to the natural manner in which the computational mesh maps to the SIMT architecture. In a typical implementation, chunks of the mesh are handed out by the CPU to groups of threads executing on the GPU, with each thread responsible for updating a single computational cell using data in nearby memory locations. Each cell update involves many instructions (some combination of arithmetic operations and memory reads/writes), and the GPU achieves its performance advantage over the CPU through a combination of parallel execution (hundreds of threads may execute instructions simultaneously on multiple GPU "cores") and latency hiding (if a thread stalls while executing a slow instruction, such as accessing off-chip memory, execution may be switched to one of thousands other "active threads" ready for execution).

While the speedups of GPU-accelerated finite volume fluid codes reported in the literature are impressive (e.g., Schive et al. [8] report a speedup of ~100x on a uniform mesh with several million cells), GPUs still have some significant limitations. The primary appeal of GPU computing is the promise of achieving the computational equivalent of 10-100 latest-generation CPU cores with a single graphics card for a fraction of the cost. Unfortunately, limited off-chip memory, absence of cache memory and poor double precision performance (or no double precision capability at all) on inexpensive consumer-grade graphics cards present barriers for the computational scientist interested in accelerating even a moderately sized (say, several million computational cells) CFD problem. While NVIDIA's Tesla cards – designed with high performance computing applications in mind – remove some of these constraints, they are significantly more expensive than the consumer-grade products (e.g., compare the ~$2500 price tag of the Tesla M2090 to the several hundred dollar cost of the GTX 480).

GPU programming effort presents another obstacle for the average computational scientist. While CUDA C has made programming graphics cards much less esoteric, taking full advantage of optimization opportunities is not a trivial exercise and requires more than a cursory knowledge of the underlying hardware. One must consider whether the investment of several programmer-months (or more) is worth the expected performance gain if one already has a code that scales well up to several thousand CPU cores and/or takes advantage of ex-

*Corresponding author. Tel: +1 301 286 9753.
**Corresponding author. Tel: +1 301 286 9753.
Email addresses: bard@astro.wisc.edu (Christopher M. Bard), john.dorelli@nasa.gov (John C. Dorelli)

isting APIs such as OpenMP and OpenACC to gain moderate speedups with relatively small amounts of programming effort. For example, GPU-accelerated finite volume solvers that use MPI to exchange ghost cell data residing on separate compute nodes may not scale gracefully up to hundreds of GPUs, since the slow rate of data transfer from GPU to CPU across the PCI bus essentially eliminates the benefit of a fast interconnect like InfiniBand. This drawback may be largely overcome, however, by NVIDIA's GPUDirect technology, which will support Direct Memory Access (DMA) between GPUs in separate compute nodes.

In this article, we describe our effort to port a MUSCL-Hancock ideal magnetohydrodynamics (MHD) solver to a single NVIDIA GTX 480 graphics card. We implement a relatively simple method for GPU speedup that does not require significant knowledge of the underlying GPU hardware. Our goal is to demonstrate that despite the difficulties inherent in GPU programming, one can achieve, with minimal programming effort, a significant performance gain (more than two orders of magnitude compared to the sequential algorithm running on a single Nehalem core) for a problem of moderate size (more than a million computational cells) on a consumer-grade graphics card. We hope that our article will serve as a roadmap for those seeking to port a standard finite volume MHD code to a GPU (or GPU cluster).

2. GTX 480 Architecture and the CUDA Programming Model

We begin with a short description of NVIDIA's Fermi architecture and the CUDA programming model (see NVIDIA White Paper [9]) for further details). The GTX 480 was the natural choice for our experiment since it was the first widely available (for several hundred USD) consumer graphics card to implement NVIDIA's Fermi architecture, which is the same architecture implemented in the much more expensive high end Tesla series marketed to the high performance computing community. The larger number of streaming processors (SP) and active threads per SP on Fermi GPUs allows for greater parallelism and latency hiding. Additionally, Fermi GPUs have improved memory performance, including L1 and L2 caches, reducing the need to explicitly stage data in the faster shared memory.

The GTX 480 consists of 15 Streaming Multiprocessors, 6 GB of GDDR5 RAM (memory bandwidth of 177.4 GB/s), a 768 KB L2 cache, a host interface (through which the GPU communicates with the host CPU via PCI-express), and a scheduler that distributes blocks of threads to the SMs. Figure 1 shows a schematic of a single Streaming Multiprocessor (SM) on the GTX 480. Each SM consists of 32 Stream Processors (SP) (what NVIDIA refers to as "CUDA cores"), 16 load/store units and 4 special function units. The SM schedules threads for execution in groups of 32 called "warps", with each thread in a warp executing the same instruction. In the Fermi architecture, there are two such "warp schedulers" per SM, each capable of issuing a single-precision instruction (e.g., a multiply or add) to 16 of the 32 cores in one shader clock

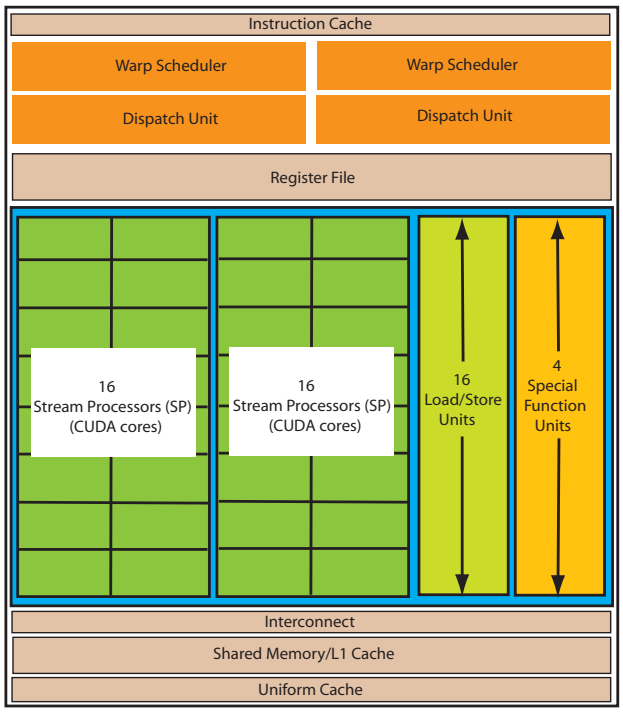


Figure 1: A GTX 480 Streaming Multiprocessor consists of two warp schedulers, each of which can issue a single-precision instruction to 16 cores every shader clock cycle; thus, the single-precision instruction throughput is one warp (32 threads) instruction every shader cycle. The double precision instruction throughput is a factor of eight slower: 1 warp instruction every 8 shader clock cycles.

cycle, thus providing a single-precision throughput of 1 warp instruction per cycle. For double-precision instructions, a single warp scheduler issues the instruction to 16 of the 32 cores in one shader clock cycle. While the hardware is capable of providing a double-precision throughput of 1 warp instruction every two cycles, NVIDIA has intentionally reduced the double precision throughput of its consumer grade cards (such as the GTX480) by a further factor of 4. With 15 SMs and a shader clock speed of 1401 MHz, the overall floating point speed of the GTX 480 is 672 GFLOPS for single-precision multiplies or adds and 84 GFLOPS for double precision multiplies or adds (double these throughputs for fused multiply-adds).

We turn now to a summary of the CUDA programming model, illustrated schematically in Figure 2 (see the NVIDIA CUDA C Programming Guide [10] for further detail). The main serial code, running on the CPU (host), allocates GPU memory and launches a kernel that executes on the GPU (device). The programmer specifies, using a simple syntactical extension of the familiar C function call, the number of threads to execute on the GPU by specifying an MxN grid of thread blocks (yellow squares in Figure 2). Note that thread blocks are a programmer-defined construct to organize the calculation on the GPU. When a SM recieves a thread block, it partitions the block into warps which are then given to the scheduler for execution. Threads within a block can access a small pool (configurable to either 16 KB or 48 KB per block in Fermi) of shared memory and can be barrier-synchronized. Thread blocks are executed asynchronously on the SMs; that is, the programmer does not have the ability to synchronize the execution of the threads in different blocks (for example, by calling a "barrier" function). This asynchronous block execution allows the same CUDA C source code to work on later-generation GPUs that can execute more blocks concurrently. However, the ability to synchronize threads within a block resulted in the design constraint (to avoid long wait times) that all threads in a block are assigned to the same SM.

NVIDIA categorizes its GPUs according to "compute capability." The GTX 480 has a compute capability of 2.0, which implies the following resource constraints:

1. The maximum number of threads per block is 1024.
2. The maximum number of blocks that can be assigned to a single SM is 8.
3. The maximum number of warps that can be managed on a single SM is 48 (implying that the maximum number of threads on an SM is 1536).

When calling a kernel function, the programmer must decide carefully how many threads to assign to a block. A general rule of thumb is that if one's application is memory bandwidth-bound, then one should strive for maximum thread occupancy (i.e., maximize the number of active warps on an SM) to take advantage of memory latency hiding. One should also set the number of threads per block to be an integer number of warps to avoid underpopulated warps (which result in idle execution resources). While NVIDIA provides an Occupancy Calculator spreadsheet to facilitate experimentation with different thread block configurations, using the Occupancy Calculator does not, as we will see below, guarantee optimal speedup.

3. Implementation of the MUSCL-Hancock scheme for ideal MHD

Implementing a finite volume fluid solver in CUDA is straightforward in principle, since the basic components of a standard Godunov scheme (reconstruction of edge states, evaluation of edge numerical fluxes using a Riemann solver, and conservative evolution of cell-centered quantities) map naturally to the CUDA SIMT architecture. That is, the components of the algorithm can be implemented as CUDA kernels in which each execution thread performs the same sequence of operations on a single computational cell. In the case of the MUSCL-Hancock scheme [11], we define a kernel for each of the following steps: 1) **slopes**: compute limited slopes at each computational cell, 2) **reconstruction**: reconstruct the conserved variables at the cell edges, 3) **evolve1**: advance reconstructed edge variables half a time step using the physical fluxes at the edges, 4) **riemann**: compute the numerical fluxes at the cell edges using a Riemann solver, and 5) **evolve2**: advance the conserved variables a full time step using the numerical fluxes computed in the previous step.

Following Dedner et al. [12], we solve the "Generalized Lagrangian Multiplier" (GLM) formulation of the MHD equations to preserve the solenoidal constraint on the magnetic field:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \quad (1)$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot \left[\rho \mathbf{v}\mathbf{v} + (p + \frac{B^2}{2})\mathbb{I} - \mathbf{B}\mathbf{B} \right] = 0, \quad (2)$$

$$\frac{\partial \mathcal{E}}{\partial t} + \nabla \cdot \left[(\frac{\rho v^2}{2} + \frac{\gamma}{\gamma - 1} p + B^2)\mathbf{v} - (\mathbf{v} \cdot \mathbf{B})\mathbf{B} \right] = 0, \quad (3)$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) + \nabla \psi = 0, \quad (4)$$

$$\frac{\partial \psi}{\partial t} + c_h^2 \nabla \cdot \mathbf{B} = -\frac{c_h^2}{c_p^2} \psi \quad (5)$$

where ρ is the mass density, $\mathbf{v}$ is the bulk velocity, p is the plasma pressure, $\mathbf{B}$ is the magnetic field, $\mathcal{E} = \rho v^2/2 + p/(\gamma - 1) + B^2/2$ is the total energy density, and γ is the ratio of specific heats (taken to be 5/3 in all of our simulations).

In equations (4) and (5), ψ is a scalar function whose evolution, is, by construction, equivalent to that of $\nabla \cdot \mathbf{B}$; thus, the parameters c_h and c_p represent the propagation and dissipation speeds of local magnetic field divergence errors. c_h is chosen to be the global maximum (over the grid cells) of $\max(|c_f + v_x|, |c_f + v_y|, |c_f + v_z|)$, where c_f is the fast magnetosonic wave speed. Note that in the 1D case (where x is the dependent variable) the equations for B_x and ψ are decoupled from the remaining MHD equations; thus, in the two-dimensional case, the globally large c_h applies only to those x numerical fluxes associated with B_x and ψ, both of which will typically be very small. Similarly, only the B_y and ψ numerical fluxes use the

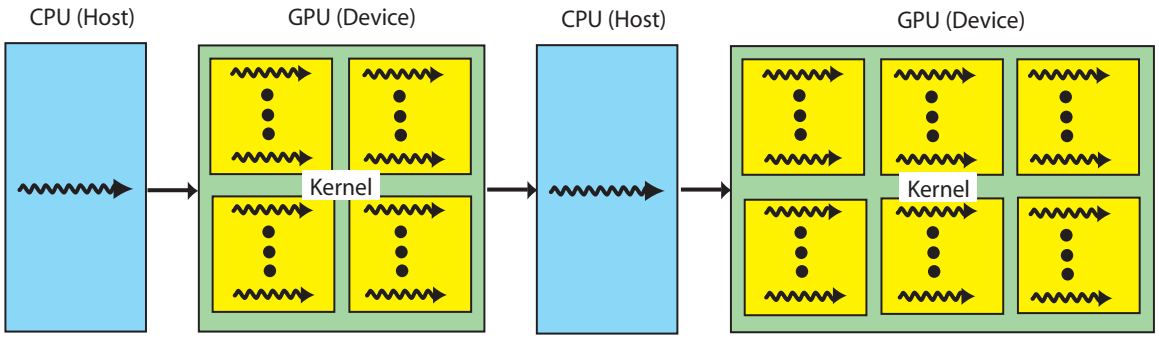


Figure 2: In the CUDA programming model, the CPU ("Host") launches a sequence of "kernels" on the GPU ("Device"). Each kernel invocation launches a user-specified number of thread blocks (yellow boxes), each block containing the same number of threads. The number of thread blocks and the number of threads/block varies from kernel to kernel, but one should in general strive to maximize the number of total active threads per SM.

globally large c_h. The remaining numerical fluxes are computed using local wave speeds. Dedner et al. [12] found that setting $c_p^2/c_h = 0.18$ produced optimal results regardless of the grid resolution, and we use the same value in all of our simulations.

We discretize the system (1)-(5) using a second-order MUSCL-Hancock scheme [11] with a minmod slope limiter to suppress oscillations near discontinuities, Strang splitting [13] to maintain second-order temporal accuracy and an HLL approximate Riemann solver [14] to compute numerical fluxes. We emphasize that our use of time-splitting was not motivated by any GPU hardware or software constraints; one could easily implement an unsplit time stepping scheme using the same basic approach (indeed, we have developed an unsplit version of this MHD code and intial tests show similar performance gains). Similarly, our choice of divergence cleaning is not driven by GPU hardware or software constraints. Other MHD algorithms which use staggered-mesh formulations to enforce the divergence constraint to machine precision [15, 16, 17, 18, 19, 20, 21] should also be easily ported to GPUs.

CUDA C source code for the main driver program (`main.cu`) and the subroutine that handles a partial time step in the X dimension (`evolvex`) are given in the Appendix. The rest of the source code is available online with the supplementary materials.

Appendix B shows the implementation of the CUDA kernel `evolvex` (the z operator is implemented in a similar manner). `evolvex` takes as its first argument the pointer `p_cons` – allocated in `main.cu` using `cudaMalloc` (see Appendix A) – to the GPU global memory where the solution vector is stored. The thread block configuration is specified by declaring two variables of type `dim3`, `dimGrid` and `dimBlock`, that specify the dimensions of our two-dimensional block of threads. (Note that the absence of a third argument in these declarations defines the third dimension to be 1.) Recalling that the number of threads in a block should be an integer multiple of 32 (to avoid underpopulated warps), we specify a `TILE_WIDTH` × `TILE_WIDTH` array of threads, where `TILE_WIDTH` is typically set to either 8 or 16 (giving, respectively, 64 and 256 threads per block; see Section 5 for a comparison of different values of `TILE_WIDTH`). The declaration of `dimGrid` specifies the dimensions of our two-dimensional grid of thread blocks. Here, we set the number of blocks in each dimension so that the total number of threads is equal to the number of computational cells (`NX` × `NZ`); thus, each thread is responsible for updating a single computational cell at each time step. We define `dimGrid` and `dimBlock` the same way throughout the code (with the exception of the boundary condition kernels, which only involve those cells at the edges). The function calls in `evolvex` map straightforwardly to the steps of the MUSCL-Hancock algorithm:

1. **slopes**: `compute_xslopes`
2. **reconstruction**: `d_from_slope`
3. **evolve1**: `d_from_fg_ker`
4. **riemann**: `wave_speeds_ker`, `physical_flux_f_ker`, `hll_flux_f`
5. **evolve2**: `cons_from_f_ker`, `damp_psic_ker`

Given a cell-averaged solution vector U_i^n, where i is the cell index and n is the time index, the first step in the MUSCL-Hancock scheme is to linearly extrapolate U_i^n to the left/right (or bottom/top) cell interfaces:

$$\begin{aligned} U_i^L &= U_i^n - \Delta U_i^n \\ U_i^R &= U_i^n + \Delta U_i^n \,, \end{aligned} \tag{6}$$

where ΔU_i^n is the slope calculated by the minmod slope limiter,

$$\Delta U_i^n = \texttt{minmod}[(U_{i+1}^n - U_i^n), (U_i^n - U_{i-1}^n)] \,. \tag{7}$$

We implement **slopes** step (7) in the kernel `compute_xslope_ker` (the kernel `compute_zslope_ker` extrapolates to the bottom/top cell interfaces). `compute_xslope` takes as input the GPU pointer to the solution vector (the vector of conserved cell

averages, p_cons) and the GPU pointer to the slopes array (p_xslope). Within compute_xslope_ker, each thread defines its location in the grid (stored in the variables a and b) using both its location within the thread block and the block's location within the grid. Each thread calculates the slope at its cell by accessing the global memory pointed to by p_cons (where index3 is a macro defined in macros.h). Slopes are then stored in the global memory pointed to by p_xslope. The remaining steps in the MUSCL-Hancock algorithm are implemented in a similar manner, with each thread responsible for its own computational cell and all executing threads performing the same sequence of operations at each time step.

We note here that our implementation of the minmod limiter involves a number of conditional statements (to check, for example, whether the left and right slopes differ in sign) which, while not optimal in CUDA's SIMT architecture, is nonetheless easier to code. Specifically, if some threads in a warp satisfy the if block and the rest satisfy the else block, then *all* threads in the warp must execute the if and else blocks in succession. We point out that other Riemann solvers, e.g. Roe-type [22], HLLC [23, 24] and HLLD [25], incorporate several "if-then" conditions which will result in a performance hit. However, we do not anticipate this to be a large effect since most neighboring cells away from a discontinuity will have similar states. This means that each thread in a warp will more often than not fulfill the same if-then condition within the Riemann solver, reducing the number of instructions each warp has to execute. In general, though, one should strive to minimize the use of conditionals with divergent threads.

The **reconstruction** step (eq. 6) is implemented in the kernel d_from_slope_ker, which is invoked in both evolvex and evolvez. d_from_slope_ker makes use of registers to limit the number of global memory accesses; the variables cons_value and slope_value store the value found in global memory, which is then used in the subsequent calculation. While reading this data from global memory and storing it in registers doubles the number of steps in the kernel, replacing the global memory reads with much faster register reads results in a speed-up of $\approx$ 15%.

After **reconstruction**, **evolve1** advances the extrapolated edge states by half a time step:

$$\begin{aligned} U_i^{R,n+1/2} &= U_i^R - \frac{\Delta t}{2\Delta x}[F(U_i^R) - F(U_i^L)] \\ U_i^{L,n+1/2} &= U_i^L - \frac{\Delta t}{2\Delta x}[F(U_i^R) - F(U_i^L)], \end{aligned} \tag{8}$$

where $F(U_i^R)$ and $F(U_i^L)$ are the ideal MHD physical fluxes evaluated at the left and right interfaces, respectively. These physical fluxes are computed in the kernel physical_flux_f_ker (likewise physical_flux_g_ker) which, like d_from_slope, makes use of thread registers to limit global memory accesses. **evolve1** (eq 8) is performed in d_from_fg_ker, which is very similar in implementation to d_from_slope_ker.

Numerical fluxes at the cell interfaces are computed using the HLL Riemann solver:

$$FU_{i+1/2}^n = \begin{cases} F(U_i^R,) & \text{if } 0 \le c_i^L. \\ \frac{c_{i+1}^R F(U_i^R) - c_i^L F(U_{i+1}^L) + c_i^L c_{i+1}^R (U_{i+1}^L - U_i^R)}{c_{i+1}^R - c_i^L}, & \text{if } c_i^L \le 0 \le c_{i+1}^R. \\ F(U_{i+1}^L), & \text{if } 0 \ge c_{i+1}^R. \end{cases} \tag{9}$$

where $i + 1/2$ denotes the interface between cells i and $i + 1$, c_i^L is the minimum wave speed in cell i, and c_{i+1}^R is the maximum wave speed in cell $i + 1$. The wave speeds are computed in kernels wave_speeds_ker and hll_flux_f_ker.

Finally, **evolve2**, implemented in kernel cons_from_f_ker, advances the cell-averaged solution vector by a full time step Δt:

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2}^n - F_{i-1/2}^n). \tag{10}$$

As a general rule, one should strive to minimize data transfers from CPU to GPU to avoid a significant PCI bandwidth bottleneck. For example, our CUDA implementation of MUSCL-Hancock involves defining a separate kernel to implement each basic component of the algorithm (**slopes**, **reconstruction**, **evolve1**, **riemann**, and **evolve2**). Thus, each kernel must receive input data from the previous kernel and send output data to the subsequent kernel. Rather than copy this data back and forth from CPU to GPU (using, for example, cudaMemcpy), it is much more efficient to allocate and deallocate GPU memory as needed (using cudaMalloc) and pass the associated pointers to the kernels. This approach requires that the entire problem live on the GPU for the duration of the calculation, thus limiting the problem size. Increasing the problem size would require running on multiple GPUs (for example, by combining CUDA and MPI), taking care to minimize communication between GPUs on different compute nodes. The availablity of large GPU clusters (e.g. Keeneland and Blue Waters) thus holds the promise of similar efficiency gains with much larger problem sizes in the weak scaling limit.

A second generic optimization strategy is to favor register and shared memory over global memory to the extent possible, taking advantage of the much lower register and shared memory latency. One must, however, be aware of an important tradeoff between register and shared memory use and thread parallelism. Register and shared memory on a particular SM is divided up among all of the thread blocks on the SM. Exceeding the register or shared memory available on the SM may greatly reduce the number of active threads since threads can only be assigned to the SM at the block granularity. Experimentation (on an application by application basis) is required to determine the proper balance between register and shared memory use, on the one hand, and thread parallelism on the other. For example, although the GTX 480 can accommodate 1024 threads per block, we found that the register footprint for our MUSCL-Hancock implementation limited us to a maximum of 256 threads per block. Interestingly, for problem sizes ranging from 64^2 to 1024^2 computational cells, 64 threads per block seemed to be optimal (for a fixed register footprint). Experimentation will be required for different architectures; the best

combination of parameters for one particular type of GPU may not be ideal for other GPUs.

4. Results

On the CPU, the total time, T_{CPU}, required to apply a single instruction over the entire computational mesh is simply the execution latency of the instruction, $L_{E,CPU}$, multiplied by the number of mesh cells, N_{cells}: $T_{CPU} = N_{cells}L_{E,CPU}$. GPUs achieve their performance gains by executing thousands of threads in parallel, thereby hiding execution latency (which in memory bound applications is dominated by global memory latency). That is, a warp scheduler on an SM issues an instruction to an active warp (i.e., a warp that is ready to receive the instruction), which then proceeds to make use of whatever SM resources (arithmetic logic units, load/store units, special function units, etc.) it needs to execute the instruction. As the warp executes its instruction, the warp scheduler attempts to issue another instruction to another active warp; latency is completely hidden when there is always an active warp available to carry out the next instruction. Note, however, that it takes a finite time – the *issue latency*, $L_{I,GPU}$ – for the scheduler to issue one instruction to a warp. Here, $L_{I,GPU}$ is defined as the inverse of the instruction throughput (the number of instructions the SM can issue per clock cycle). For example, as we reviewed in section 2, the single-precision instruction throughput for the GTX 480 is 1 instruction per shader clock cycle so that $L_{I,GPU} = 1$ cycle (1/1401 μsec for the GTX 480). If $L_{E,GPU}$ is the execution latency of the instruction (i.e., the time it takes a warp to complete the instruction), then, assuming that there is always an active warp ready to receive the next instruction (memory latency is completely hidden), the total time, T_W, for the N_W active warps on an SM to complete the instruction is simply $T_W = (N_W - 1)L_{I,GPU} + L_{E,GPU}$. Since each active thread is responsible for a single computational cell, the N_{SM} SMs in our GPU can process $32N_WN_{SM}$ mesh cells in time T_W; thus, it takes the GPU a time $T_{GPU} = T_W[N_{cells}/(32N_WN_{SM})]$ to process all N_{cells} cells in our computational mesh with the same instruction. The expected speedup, S, for a sequence of add instructions (assuming the maximal number of active threads on the GPU) can thus be estimated as follows:

$$S = \frac{T_{CPU}}{T_{GPU}} = \frac{32N_WN_{SM}L_{E,CPU}}{L_{E,GPU} + (N_W - 1)L_{I,GPU}} \qquad (11)$$

As an example, let us estimate the expected speedup for a sequence of double precision adds (see the source code in Appendix E and Appendix F). Timing of the CPU code running on a single core (using gcc with the "-O3" option) resulted in an average execution latency, $L_{E,CPU}$, of approximately 10 CPU cycles for our 2.8 GHz Intel Nehalem (with array sizes of 1024x1024). Assuming $N_W = 48$ (full occupancy), $L_{I,GPU} = 8$ shader cycles (recall that NVIDIA reduced the double precision throughput of the GTX 480 by a factor of 4 below the 1/2 double precision instruction per shader cycle of the HPC-grade Fermi cards), and $L_{E,GPU} \approx 600$ shader clock cycles (the GTX 480 global memory latency) in (11), we obtain an overall speedup $S \approx 118$ compared to the sequential algorithm running on a single CPU core. This is close to our maximum speedup of 126 reported in Table 2. Timing of the corresponding CUDA kernel in Appendix F gives a speedup $S \approx 112$.

We emphasize that our simple estimate does not take into account other factors that influence GPU and CPU performance. Specifically, (11) is independent of problem size; nevertheless, we expect the CPU memory latency to increase substantially for larger problems due to cache misses. Since memory latency is effectively hidden by the GPU, we expect more impressive GPU speedups for larger problem sizes. Additionally, we do not take into account other issues such as the relatively slow transfer of data between the CPU and GPU or the granularity of thread block sizes. Our results show that the speedups do depend on the number of threads per block, which indicates that the block size affects the number of active warps per SM. This would change the value of N_W in the above speedup equation. These are likely the biggest factors accounting for the difference between the estimated speedup (using (11)) and our measured speedups (Table 2). Finally, to be fair, we should compare our GPU performance on a parallel implementation of the CPU application that makes use of all four Nehalem cores; in this case, we expect an overall speedup – relative to a parallelized version of our MUSCL-Hancock code running on all four CPU cores – of roughly $118/4 \approx 30$. Additionally, many production machines have CPUs with 8 to 16 cores; this would result in a speedup closer to 10.

Ideally, one would like to achieve speedups consistent with those predicted by equation (11) for a "real world" application (e.g., our MUSCL-Hancock algorithm) involving multiple kernels executing sequences of instructions consisting of many operations (not limited to adds and multiplies). To measure the performance of our GPU-accelerated MUSCL-Hancock solver, we simulated the Orszag-Tang vortex with the following initial conditions (see, for example, Dai and Woodward [17]): $\rho = 25/(36\pi)$, $p = 5/(12\pi)$, $v_x = -\sin(2\pi z)$, $v_y = 0.0$, $v_z = \sin(2\pi x)$, $b_x = \sin(2\pi z)/(4\pi)$, $b_y = 0.0$, and $b_z = \sin(2\pi x)/(4\pi)$. The simulation was run for 300 time steps in all test cases (Figure 3 shows the simulation results for a longer simulation that ran for approximately two Alfvén times). The CPU C source used in the test was compiled using gcc-4.4.3 for two cases: 1) with "-O3" option enabled (hereafter referred to as the "Optimized CPU", and 2) without the "-O3" option enabled (hereafter referred to as "Unoptimized CPU"). The simulation was run on a single Intel Core i7 930 (2.8 GHz) core. The CPU wall clock time was compared to the wall clock time of several versions of the CUDA GPU code running on the GTX 480 (see Tables 2 and 3): 1) "Register..." refers to versions with `TILE_WIDTH` = {2, 4, 8, 16} that make use of register memory wherever possible to store intermediate values (the solution vector, however, remains in GPU global memory for the entire calculation); 2) "Global..." refers to the CUDA version that makes extensive use of GPU global memory with minimal usage of registers and `TILE_WIDTH` = {2, 4, 8, 16}. The timing results included disk I/O time; that is, we started our clock at the beginning of the main program and ended the clock after the final output had been written to disk.

Problem Size	Unoptimized C	Optimized C	CUDA
64^2	13.37 s	6.45 s	0.57 s
128^2	73.39	41.80	1.81
256^2	484.33	277.73	5.24
512^2	2366.45	1476.98	18.27
1024^2	11488.6	8029.35	63.84

Table 1: Average run times for various problem sizes for an unoptimized ideal MHD C code, an optimized ideal MHD C code, and a CUDA ideal MHD code. The CUDA code used here is identical to the `TILE_WIDTH` 8 code in Table 2.

Problem Size	Register 16	Register 8	Register 4	Register 2
64^2	0.8 s (8.1)	0.57 s (11)	0.67 s (9.7)	1.2 s (5.4)
128^2	2.25 (19)	1.81 (23)	2.04 (21)	4.34 (9.6)
256^2	6.52 (43)	5.24 (53)	6.71 (41)	16.13 (17)
512^2	22.58 (65)	18.72 (81)	25.19 (59)	68.12 (22)
1024^2	84.62 (95)	63.84 (126)	90.61 (89)	253.88 (32)

Table 2: Average run times for permutations of our single GPU ideal MHD code. Register *number* refers to the codes using a register-heavy approach with the `TILE_WIDTH` set to *number*. Numbers in parentheses are the approximate speedups relative to the Optimized C timings in Table 1.

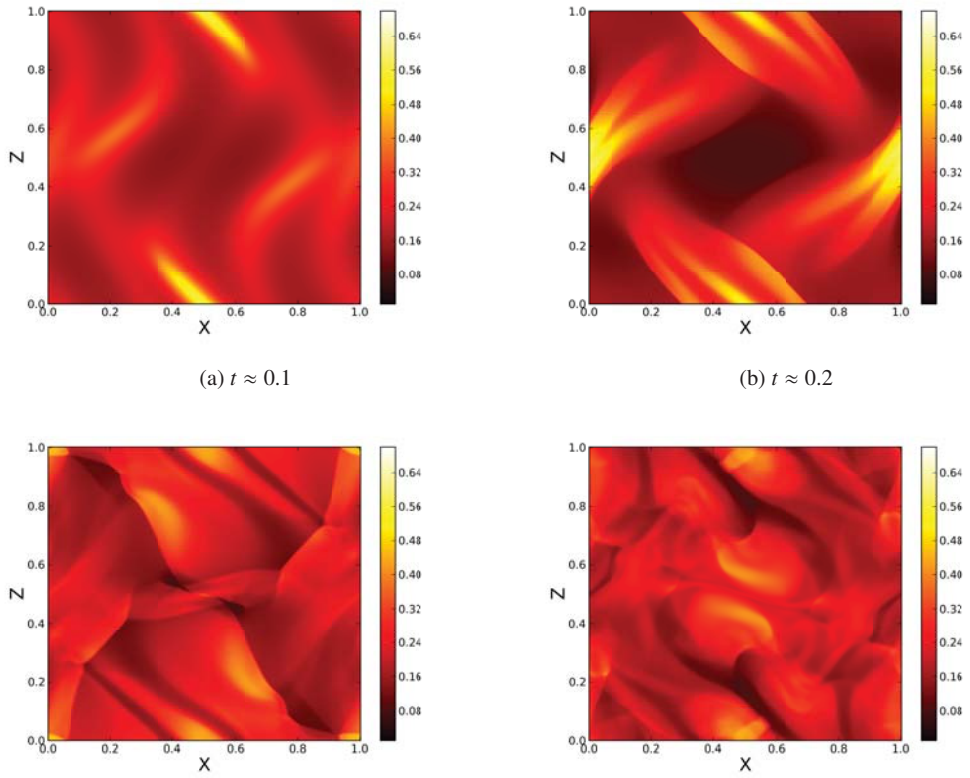
On average, the CUDA code is roughly 59 times faster than Optimized C; however, it is clear from Table 1 that the speedup increases with problem size. For example, for a 64^2 double precision problem, the CUDA code executes only 11 times faster than Optimized C, while the 1024^2 double precision CUDA code is 126 times faster than Optimized C. The increase in speedup with problem size likely reflects the fact that the CPU code is memory bound for large problem sizes, so that memory bandwidth and latency determine the CPU execution latency, $L_{E,CPU}$ in equation (11); in contrast, the global memory latency is effectively hidden for large problems running on the GPU. Note that our CPU code does not implement any special optimizations to overcome the memory bottleneck, e.g. strip mining [26], space tiling [27], or linear loop transformations [28].

It is clear from Tables 2 and 3 that `TILE_WIDTH` = 8 seems to be the "sweet spot" where the number of threads per block is large enough to effectively hide global memory latency but not so large that heavy use of thread registers reduces SM occupancy (due to block granularity of thread assignments). From our testing, the register-heavy code is always faster (by $\approx$ 15 – 25%) than the global memory-heavy codes, keeping `TILE_WIDTH` constant. Interestingly, the Global 8 code turns out to be faster than the Register 16 code. Our finding that using 64 threads per block results in the fastest code is puzzling, given that the NVIDIA CUDA Occupancy Calculator suggests that our application can accommodate 256 threads/block (which should, in principle, result in a greater degree of parallelism and memory latency hiding). Ultimately, these results emphasize the importance of experimentation to find the right balance of register use and number of threads per block.

5. Conclusions and Discussion

In this paper, we have described in detail our experience porting a two-dimensional MUSCL-Hancock ideal MHD solver to a single NVIDIA GTX 480 (Fermi architecture) using CUDA C. We compared a sequential CPU version of the algorithm – compiled with gcc-4.4.3 and executed on a single Intel Core i7 930 (2.8 GHz) CPU core – to a parallel version running on a single GTX 480. We made no special effort to stage data in the much faster shared memory of the GPU; rather, we experimented with different combinations of thread register use and thread block size to maximize the number of active threads per Streaming Multiprocessor (SM) (and thereby hide memory latency for our memory bound application). For a double precision problem with 1024^2 mesh cells, we achieved a maximum speedup of roughly 126 (with the CPU source code compiled using the "-O3" gcc optimization flag) by making use of thread registers to store kernel variables whenever possible and setting the number of threads per block to 64. Interestingly, keeping all of the data in global memory (with 64 threads per block) was more efficient than using large amounts of register memory in combination with a larger number (256) of threads per block.

Several other groups (see, for example, Schive et al. [8], Wong et al. [29] and references therein) have reported on GPU implementations of MHD codes. The results of Wong et al. [29] (hereafter W11) are particularly relevant to our work, since their algorithm is an explicit finite volume method on a uniform mesh (in contrast to the AMR simulation tested by Schive et al. [8]), and their timing tests included a two-dimensional case on a GTX 480. For problem sizes of 512^2 and 1024^2, W11 quote speedups (relative to a single 3.2 GHz Intel Nehalem core) of 320 and 600, respectively. For a 128^3 double precision problem (twice the size of their 1024^2 problem with speedup of 600), W11 report a speedup of only 155. They report a speedup of



(a) $t \approx 0.1$

(b) $t \approx 0.2$

(c) $t \approx 0.4$

(d) $t \approx 0.8$

Figure 3: Evolution of the Orszag-Tang vortex for a grid size of 512^2.

260 for a 64^3 problem (still less than the reported 320 speedup for a 2D problem with the same number of computational cells). These speedups are higher than ours for similarly sized problems, and none are consistent with the simple estimate based on equation (11).

We speculate that the discrepancy between our speedups and those reported by W11 is a result of inefficient memory access in the 1D and 2D versions of the Fortran code they used for their CPU runs. Specifically, it is clear from Table 4 of W11 that the $\mathbf{fluid}_y$ and $\mathbf{fluid}_z$ operations are much more expensive than $\mathbf{fluid}_x$ (due to less efficient memory access for the y and z fluid updates), whereas these operations take similar time on the GTX 480 (where memory latency is effectively hidden). Further, W11's Fortran CPU code calls the **Transposition** operation, whereas their CUDA code does not. Similarly, Table 5 of W11 shows that $\mathbf{fluid}_z$ is much more expensive than $\mathbf{fluid}_x$ or $\mathbf{fluid}_y$ (which are comparable since the **Transposition** operation allows efficient memory access in the x and y dimensions) in the Fortran CPU code, while all three operations take similar times in the CUDA code. In contrast, $\mathbf{fluid}_x$, $\mathbf{fluid}_y$ and $\mathbf{fluid}_z$ all take similar times in the 3D cases (for which the **Transposition** operation allows data in all three dimensions to be accessed efficiently).

Thus, it appears that W11's CPU implementation uses a fluid update algorithm which is significantly less efficient than the GPU algorithm for 1D and 2D problems. In the 64^3 case reported in their Table 6, if we ignore the **Transposition** operation (not included in W11's GPU implementation), their GTX 480 single precision speedup is 97 for a problem size of 64^3. Using W11's reported (in their Table 3) double vs. single precision run time of 1.3973 for their 64^3 problem, their double precision speedup is roughly 69, which is much closer to the double precision speedup (over the optimized C code, compiled with the "-O3" gcc option) of 67 we observe for our "Global 8" 512^2 problem (with the same number of computational cells) in Table 3. Thus, we are confident that the speedups we report in this paper are consistent with those reported by W11.

Though these results are encouraging for applications that fit on a single GPU, real world applications are much larger than 1024^2 and generally must be run on thousands of compute nodes. The newest supercomputers (e.g. Keeneland, Blue Waters, Titan) are being created with a hybrid architecture in which each node consists of a multi-core CPU and one or more GPUs. The method presented here is chiefly based on minimizing the amount of data being transfered between the CPU and GPU, so the natural extension to many nodes is to minimize both the intra- and inter-node communication. In our approach, each thread on the GPU is responsible for one computational

Problem Size	Global 16	Global 8	Global 4	Global 2
64^2	0.95 s (6.8)	0.63 s (10)	0.82 s (7.9)	1.59 s (4)
128^2	2.74 (15)	2.09 (20)	2.52 (17)	5.93 (7)
256^2	8.51 (33)	6.16 (45)	8.76 (32)	22.825 (12)
512^2	29.92 (49)	22.06 (67)	33.36 (44)	97.46 (15)
1024^2	113.88 (71)	77.34 (104)	120.31 (67)	357.69 (22)

Table 3: Average run times for permutations of our single GPU ideal MHD code. Global *number* refers to the codes using a global memory-heavy approach with the `TILE_WIDTH` set to *number*. Numbers in parentheses are the approximate speedups relative to the Optimized C timings in Table 1.

cell; this allows us to divide the overall simulation space among multiple GPUs since individual cell calculations are independent from other cell calculations. In this case, the only inter-GPU communication required is the boundary cell states prior to the time evolution and the timestep. Currently, passing data between GPUs in different compute nodes represents a significant communication bottleneck. We note that with CUDA 5, however, NVIDIA has implemented "GPUDirect", which enables direct communication between GPUs in a cluster rather than routing the data through their associated CPUs with MPI.

Though we have presented results indicating that a single-GPU code shows significant speedup over a single-CPU code, it is not yet clear whether a multi-GPU code will offer the same speedup relative to a multi-CPU code. It is possible, however, that an explicit finite volume MHD solver might achieve good weak scaling up to thousands of GPUs using a message passing approach in which only ghost cell data is communicated between GPUs (with the rest of the solution remaining on the GPUs for the entire calculation).

In summary, we have demonstrated that porting a non-trivial finite-volume algorithm (a second-order MUSCL-Hancock ideal MHD solver) to a modern (compute capability 2.0 or higher) GPU can yield a significant speedup compared to a single Nehalem core – comparable to that predicted by a simple estimate based on the arithmetic throughput, memory latency and parallel processing capacity of the GPU – with minimal programming effort. While some experimentation with different combinations of thread register use and thread block size is useful, one can in general expect speedups of ≈ 100 for moderately large double precision problems ($> 1024^2$ computational cells). Future work will focus on comparing a multi-GPU MHD code with a similar multi-CPU MHD code in order to delineate the benefits and drawbacks of GPUs vs. CPUs in the high-performance supercomputing realm. For now, we are confident in recommending GPUs as a viable speedup tool for small- to moderate-sized simulations ($< 10^6$ cells).

Acknowledgment

The authors would like to thank X. Feng and H.C. Wong for helpful discussion regarding their timing results.

CB was supported by the Delaware Space Grant program (NASA Grant NNG05G092H) and the NASA GSRP Fellowship (NASA Grant NNX11AK70H).

Appendix A. GPU main

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cuda.h>
#include <time.h>
//list of definitions
#include "userinput.h"
#include "macros.h"
#include "functions.h"

int main(){
  int n, nsteps, nskip;
  double cfl, time, dt;
  double wspeed_max;

  FILE *output = fopen(FILEOUTPUT, "w");
  FILE *aux = fopen(FILEAUX, "w");

  double *p_cons;
  cudaMalloc((void**)&p_cons,
             NX*NZ*NVAR*sizeof(double));

  printf("initial\n");
  initialize_aux(aux);
  //write to aux file
  fprintf(aux, "%d %d\n", NX,NZ);

  //initialize variables
  initialcondition(p_cons);
  write_solution(p_cons, output);

  //time loop
  cfl = 0.4;
  time = 0.0;

  if(MOVIES != 1){
    nsteps = NTIMESTEPS;
    if(nsteps % (MOVIES-1) != 0){
      nsteps--;
    }
    nskip = nsteps/(MOVIES-1)-1;
    for(n=0; n<nsteps;n++){
      wspeed_max = get_max_speed(p_cons);
      dt = time_step(wspeed_max, cfl);
      evolvex(p_cons, dt, wspeed_max);
      evolvez(p_cons, dt, wspeed_max);
```

```
      evolvez(p_cons, dt, wspeed_max);
      evolvex(p_cons, dt, wspeed_max);
      time = time + 2.*dt;
      fprintf(aux, "%d %.10f %.10f\n",
              n+1, 2.*dt, time);
      printf("%d %f %f\n", n+1,2.*dt,time);
      if(n % (nskip+1) == 0){
        write_solution(p_cons, output);
      }
    }
  }

  printf("done\n");

  cudaFree(p_cons);
  fclose(output);
  fclose(aux);
  return 0;
}
```

Appendix B. GPU evolvex

```
void evolvex(double* p_cons, double dt,
             double ch){
  double dx = LX/(NX-2);
  int size_spd = NX*NZ*3*sizeof(double);
  int size3 = NX*NZ*NVAR*sizeof(double);
  dim3 dimGrid(NX/TILE_WIDTH, NZ/TILE_WIDTH);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

  //GPU pointers
  double *p_dright, *p_dleft, *p_fright;
  double *p_fleft, *p_fvec;
  double *p_cright_min, *p_cright_max;
  double *p_cleft_min, *p_cleft_max;
  double *p_xslope;

  cudaMalloc((void**)&p_xslope, size3);

  compute_xslope_ker<<<dimGrid, dimBlock>>>
                       (p_cons, p_xslope);

  cudaMalloc((void**)&p_dright, size3);
  cudaMalloc((void**)&p_dleft, size3);

  d_from_slope(p_dright, p_dleft,
               p_cons, p_xslope);

  cudaFree(p_xslope);

  cudaMalloc((void**)&p_fleft, size3);
  cudaMalloc((void**)&p_fright, size3);

  physical_flux_f_ker<<<dimGrid, dimBlock>>>
                     (p_dright, p_fright, ch);
  physical_flux_f_ker<<<dimGrid, dimBlock>>>
                     (p_dleft, p_fleft, ch);
  d_from_fg_ker<<<dimGrid, dimBlock>>>(p_dright,
                    p_dleft, p_fright, p_fleft,
                    dt, dx);

  cudaMalloc((void**)&p_fvec, size3);
  cudaMalloc((void**)&p_cright_max, size_spd);
  cudaMalloc((void**)&p_cright_min, size_spd);
  cudaMalloc((void**)&p_cleft_max, size_spd);
  cudaMalloc((void**)&p_cleft_min, size_spd);

  wave_speeds_ker<<<dimGrid, dimBlock>>>
                 (p_dright, p_cright_min,
                  p_cright_max, ch);
  wave_speeds_ker<<<dimGrid, dimBlock>>>
                 (p_dleft, p_cleft_min,
                  p_cleft_max, ch);
  hll_flux_f(p_dleft, p_dright, p_fleft,
             p_fright,p_fvec, p_cleft_min,
             p_cleft_max,p_cright_min,
             p_cright_max, ch);

  cudaFree(p_cleft_min);
  cudaFree(p_cleft_max);
  cudaFree(p_cright_min);
  cudaFree(p_cright_max);
  cudaFree(p_dright);
  cudaFree(p_dleft);
  cudaFree(p_fright);
  cudaFree(p_fleft);

  cons_from_f_ker<<<dimGrid, dimBlock>>>
                  (p_cons, p_fvec, dt);

  cudaFree(p_fvec);

  if(DIV_CLEAN == 1){
    damp_psic_ker<<<dimGrid, dimBlock>>>
                   (p_cons, ch, dt);
  }

  if(PROBLEM == ORSZAG_TANG){
    boundary_x(p_cons);
    boundary_z(p_cons);
  }
  if(PROBLEM == BRIO_WU){
    boundary_shock(p_cons);
  }
  return;
}
```

Appendix C. CPU main

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
//list of definitions
#include "CPU_userinput.h"
#include "CPU_macros.h"
#include "CPU_functions.h"

int main(){
  int a,b,c,n, nsteps, nskip;
  double cfl, time, dt, dx, dz;
  double wspeed_max, x, z;
  int fileno;
```

```
  double movie_dt, movie_time;

  FILE *aux = fopen(FILEAUX, "w");
  double* cons = (double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));

  //write to aux file
  fprintf(aux, "%d %d\n", NX,NZ);
  //write xgrid and zgrid to aux file
  x=0;
  dx = LX/(NX-2);
  for(a=0; a<NX;a++){
    fprintf(aux, "%.15f\n", x);
    x = x + dx;
  }
  z =0;
  dz = LZ/(NZ-2);
  for(b=0;b<NZ;b++){
    fprintf(aux, "%.15f\n", z);
    z = z+dz;
  }

  //initialize variables
  initialcondition(cons);
  write_solution(cons, 0);

  //time loop
  cfl = 0.2;
  time = 0.0;
  movie_dt = TOTAL_TIME/(FRAME_RATE * MOVIE_TIME);
  movie_time = 0.0 + movie_dt;
  fileno = 1;

  for(n=1; n<=NTIMESTEPS;n++){
    wspeed_max = maximumspeed(cons)
    dt = time_step(wspeed_max, cfl);

    evolvex(cons, dt, wspeed_max);
    evolvez(cons, dt, wspeed_max);
    evolvez(cons, dt, wspeed_max);
    evolvex(cons, dt, wspeed_max);

    time = time + 2.*dt;
    fprintf(aux, "%d %.10f %.10f\n",
                     n+1, 2.*dt, time);

    printf("%d %f %f\n", n,2.*dt,time);
    if(time > movie_time){
      write_solution(cons, fileno);
      printf("Output written\n");
      movie_time = movie_time+movie_dt;
      fileno++;
    }
  }

  printf("done\n");
  free(cons);
  fclose(aux);
  return 0;
}
```

Appendix D. CPU evolvex

```
void evolvex(double* cons,
             double dt, double ch){
  int a,b,c;
  double dx = LX/(NX-2);
  double cp = sqrt(ch*CR);

  //wavespeed arrays
  double* cright_min=(double *)calloc(NX*NZ*3,
                                  sizeof(double));
  double* cright_max=(double *)calloc(NX*NZ*3,
                                  sizeof(double));
  double* cleft_min=(double *)calloc(NX*NZ*3,
                                  sizeof(double));
  double* cleft_max=(double *)calloc(NX*NZ*3,
                                  sizeof(double));
  //data arrays
  double* xslope=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));
  double* dright=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));
  double* dleft=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));
  double* fright=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));
  double* fleft=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));
  double* fvec=(double *)calloc(NX*NZ*NVAR,
                                  sizeof(double));

  compute_xslope(cons, xslope);

  //reconstruct interfaces
  for(c=0;c<NVAR;c++){
   for(b=0;b<NZ;b++){
    for(a=0;a<NX; a++){
      dright[index3(a,b,c)]=cons[index3(a,b,c)]
                    +0.5*xslope[index3(a,b,c)];

      dleft[index3(a,b,c)]=cons[index3(a,b,c)]
                    -0.5*xslope[index3(a,b,c)];
    }
   }
  }

  physical_flux_f(dright, fright, ch);
  physical_flux_f(dleft, fleft, ch);

  // half timestep evolution
  for(c=0;c<NVAR;c++){
   for(b=0;b<NZ;b++){
    for(a=0;a<NX; a++){
     dright[index3(a,b,c)]=dright[index3(a,b,c)]
              -dt/2.0/dx*(fright[index3(a,b,c)]
                        - fleft[index3(a,b,c)]);
     dleft[index3(a,b,c)]=dleft[index3(a,b,c)]
            -dt/2.0/dx*(fright[index3(a,b,c)]
                      -fleft[index3(a,b,c)]);
    }
   }
  }
```

```
  wave_speeds(dright, cright_min,
              cright_max, ch);
  wave_speeds(dleft, cleft_min,
              cleft_max, ch);

  hll_flux_f(dleft, dright, fvec, cleft_min,
             cleft_max, cright_min,
             cright_max, ch);

  //full timestep evolution
  for(c=0;c<NVAR;c++){
    for(b=1;b<NZ-1;b++){
      for(a=1;a<NX-1; a++){
       cons[index3(a,b,c)]=cons[index3(a,b,c)]
                   -dt/dx*(fvec[index3((a+1),b,c)]
                            -fvec[index3(a,b,c)]);
      }
    }
  }

  //implement psic damping
  for(b=1;b<NZ-1;b++){
   for(a=1;a<NX-1;a++){
     cons[index3(a,b,8)]=cons[index3(a,b,8)]
                         *exp(-dt*(ch/cp)*(ch/cp));
   }
  }

  boundary(cons);

  free(xslope);
  free(dright);
  free(dleft);
  free(fleft);
  free(fright);
  free(fvec);
  free(cright_min);
  free(cright_max);
  free(cleft_min);
  free(cleft_max);
  return;
}
```

Appendix E. CPU benchmark

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void get_walltime(double* wcTime);
void get_walltime_(double* wcTime);

void main()
{
  int SIZE = 1024*1024;
  double* a=(double*)malloc(n*sizeof(double));
  double* b=(double*)malloc(n*sizeof(double));
  double* c=(double*)malloc(n*sizeof(double));
  double* d=(double*)malloc(n*sizeof(double));

  int i;

  for (i = 0; i<SIZE; i++){
    a[i] = 0.;
    b[i] = 1.;
    c[i] = 2.;
    d[i] = 3.;
  }

  double start_time;
  get_walltime(&start_time);

  for (i = 0; i<SIZE; i++){
    a[i] = b[i] + c[i];
  }

  double end_time;
  get_walltime(&end_time);

  double time = end_time - start_time;
  double latency = time/(SIZE)*2.801e9;
  double mflops = SIZE/(time*1e6);
  printf("Execution time:  %.20f\n", time);
  printf("Execution latency:  %.20f\n", latency);
  printf("Processor MFLOPS:  %f\n", mflops);
}

void get_walltime_(double* wcTime)
{
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec +
                     tp.tv_usec/1000000.);
}

void get_walltime(double* wcTime)
{
  get_walltime_(wcTime);
}
```

Appendix F. GPU benchmark

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <cuda.h>
#include <cuda_runtime.h>

void get_walltime(double* wcTime);
void get_walltime_(double* wcTime);
__global__ void add_kernel(int tile_width,
                           int nx, int ny,
                           double* a,
                           double* b,
                           double* c);


int main()
{
  int nx = 1024;
  int ny = 1024;
```

```
  int tile_width = 16;
  int SIZE = nx*ny;
  int N_SM = 15;
  int N_T = 32;
  int N_W = 48;

  double *a, *b, *c;

  cudaMalloc((void**)&a, n*sizeof(double));
  cudaMalloc((void**)&b, n*sizeof(double));
  cudaMalloc((void**)&c, n*sizeof(double));

  double start_time;
  get_walltime(&start_time);

  dim3 dimGrid(nx/tile_width, ny/tile_width);
  dim3 dimBlock(tile_width, tile_width);

  add_kernel<<<dimGrid, dimBlock>>>(tile_width,
                                    nx, ny,
                                    a, b, c);


  double end_time;
  get_walltime(&end_time);

  double time = end_time - start_time;
  /* Note that this measures combined
    execution+issue latency on GPU */
  double latency=N_T*N_W*N_SM*time/SIZE*1401e6;
  double mfl = n/((end_time - start_time)*1e6);

  printf("Execution time:  %.20f\n", time);
  printf("Execution latency:  %.20f\n", latency);
  printf("Processor MFLOPS:  %f\n", mfl);

  return 0;
}

void get_walltime_(double* wcTime)
{
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec +
                     tp.tv_usec/1000000.);
}

void get_walltime(double* wcTime)
{
  get_walltime_(wcTime);
}

__global__ void add_kernel(int tile_width,
                           int nx, int ny,
                           double* a,
                           double* b,
                           double* c){

   int i = blockIdx.x*tile_width + threadIdx.x;
   int j = blockIdx.y*tile_width + threadIdx.y;

   int idx = ny*i + j;

   a[idx] = b[idx] + c[idx];
}
```

References

[1] D. B. Kirk, W. mei W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2010.

[2] R. G. Belleman, J. Bédorf, S. F. P. Zwart, High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda, New Atronomy 13 (2008).

[3] E. Gaburov, J. Bédorf, S. P. Zwart, Gravitational tree-code on graphics processing units: Implementation in cuda, Procedia Computer Science 1 (2012) 1119–1127.

[4] C. Harris, K. Haines, L. Stavely-Smith, Gpu accelerated radio astronomy signal convolution, Exp. Astron. 22 (2008) 129–141.

[5] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comput. Chem. 28 (2007) 2618–2640.

[6] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, V. S. Pande, Accelerating molecular dynamic simulation on graphics processing units, J. Comp. Chem. 30 (2009) 864–872.

[7] T. Brandvik, G. Pullan, Acceleration of a 3d euler solver using commodity graphics hardware, 2008. 46th AIAA Aerospace Sciences Meeting and Exhibit.

[8] H.-Y. Schive, Y.-C. Tsai, T. Chiueh, Gamer: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics, Astrophys. J. Suppl. 186 (2010) 457–484.

[9] NVIDIA White Paper, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Technical Report, NVIDIA Corporation, 2009.

[10] NVIDIA CUDA Documentation, CUDA C Programming Guide Version 4.2, NVIDIA Corporation, 2012.

[11] B. van Leer, On the relation between the upwind-differencing schemes of godunov, enquist-osher and roe, SIAM J. Sci. Statist. Comput. 5 (1984) 1–20.

[12] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, M. Wesenberg, Hyperbolic divergence cleaning for the mhd equations, J. Computational Phys. 175 (2002) 645–673.

[13] G. Strang, On the construction and comparison of difference schemes, SIAM J. Numerical Anal. 5 (1968) 506–517.

[14] E. F. Toro, Riemann solvers and numberical methods for fluid dynamics: A practical introduction, Springer-Verlag, 1999.

[15] C. R. Evans, J. F. Hawley, Simulation of magnetohydrodynamic flows - A constrained transport method, Astrophys. J. 332 (1988) 659–677.

[16] D. S. Balsara, D. S. Spicer, A Staggered Mesh Algorithm Using High Order Godunov Fluxes to Ensure Solenoidal Magnetic Fields in Magnetohydrodynamic Simulations, Journal of Computational Physics 149 (1999) 270–292.

[17] W. Dai, P. R. Woodward, On the divergence-free condition and conservation laws in numerical simulations for supersonic magnetohydrodynamic flows, Astrophys. J. 494 (1998) 317–355.

[18] D. Ryu, F. Miniati, T. W. Jones, A. Frank, A Divergence-free Upwind Code for Multidimensional Magnetohydrodynamic Flows, Astrophys. J. 509 (1998) 244–255.

[19] P. Londrillo, L. del Zanna, On the divergence-free condition in Godunov-type schemes for ideal magnetohydrodynamics: the upwind constrained transport method, Journal of Computational Physics 195 (2004) 17–48.

[20] J. M. Stone, T. A. Gardiner, P. Teuben, J. F. Hawley, J. B. Simon, Athena: A New Code for Astrophysical MHD, Astrophys. J., Suppl. Ser. 178 (2008) 137–177.

[21] D. Lee, A. E. Deane, An unsplit staggered mesh scheme for multidimensional magnetohydrodynamics, Journal of Computational Physics 228 (2009) 952–975.

[22] P. L. Roe, Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes, Journal of Computational Physics 43 (1981) 357–372.

[23] E. F. Toro, M. Spruce, W. Speares, Restoration of the contact surface in the HLL-Riemann solver, Shock Waves 4 (1994) 25–34.

[24] P. Batten, N. Clarke, C. Lambert, D. M. Causon, On the choice of

wavespeeds for the hllc riemann solver, SIAM J. Sci. Comput. 18 (1997) 1553–1570.

[25] T. Miyoshi, K. Kusano, A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics, Journal of Computational Physics 208 (2005) 315–344.

[26] S. Carr, K. Kennedy, Compiler blockability of numerical algorithms, in: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Supercomputing '92, IEEE Computer Society Press, 1992, pp. 114–124.

[27] M. Wolfe, Iteration space tiling for memory hierarchies, in: Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics, 1989, pp. 357–361.

[28] W. Li, K. Pingali, Access normalization: Loop restructuring for numa computers, ACM Trans. Comput. Syst. 11 (1993) 353–375.

[29] H.-C. Wong, U.-H. Wong, X. Feng, Z. Tang, Efficient magnetohydrodynamic simulations on graphics processing units with cuda, Comp. Phys. Comm. 182 (2011) 2132–2160.